Received: Added at production | Revised: Added at production | Accepted: Added at production
DOI: 10.1002/cmm4.1148

SPECIAL ISSUE PAPER

A software cache autotuning strategy for dataflow computing with UPC++ DepSpawn

Basilio B. Fraguela* | Diego Andrade

Universidade da Coruña, CITIC - Centro Singular de Investigación de Galicia , Computer Architecture Group, A Coruña, Spain

Correspondence
*Basilio B. Fraguela. Facultade de Informática, Campus de Elviña, s/n. 15071. A Coruña, Spain. Email: basilio.fraguela@udc.es

Summary

Dataflow computing allows to start computations as soon as all their dependences are satisfied. This is particularly useful in applications with irregular or complex patterns of dependences which would otherwise involve either coarse grain synchronizations which would degrade performance, or high programming costs. A recent proposal for the easy development of performant dataflow algorithms in hybrid shared/distributed memory systems is UPC++ DepSpawn. Among the many techniques it applies to provide good performance is a software cache that minimizes the communications among the processes involved. In this paper we provide the details of the implementation and operation of this cache and we present an auto-tuning strategy that simplifies its usage by freeing the user from having to estimate an adequate size for this cache. Rather, the runtime is now able to define reasonably sized caches that provide near optimal behavior.

KEYWORDS:
runtimes, auto-tuning, locality, dataflow computing, distributed memory, PGAS

1 | INTRODUCTION

Parallel applications are notoriously more difficult to develop than sequential ones, and this is particularly true when distributed memory systems are considered. This is why, beyond the traditional message-passing approach followed for decades to program these systems, researchers have explored new higher-level paradigms that simplify the exploitation of these systems while providing competitive performance. A notable proposal in this area is Partitioned Global Address Space (PGAS)[1], in which processes have both a private and a shared address space, the latter one being clearly partitioned among them so that it is easy to reason on locality and access costs. There have been many implementations of this strategy, both by means of new languages[2,3,4,5,6] and libraries integrated in popular existing languages[7,8]. In our opinion the latter strategy is the most promising one because libraries facilitate code reusability and integration with existing tools and frameworks, besides usually presenting shorter learning curves than new languages.

Although, as mentioned above, there are many PGAS implementations, which have explored a large set of synchronization strategies, the development on them of applications with complex or irregular patterns of dependences is far from easy. A very effective strategy to express algorithms of this kind is to rely on a dataflow approach that automatically respects the implicit dependencies between the tasks that are given by the data dependences among their arguments. This paradigm has repeatedly shown to largely improve the performance of critical mathematical algorithms[9,10,11], therefore being of great practical interest. While several tools support this strategy, particularly in shared memory environments[12], there were no PGAS alternative that supported it until the development of UPC++ DepSpawn[13]. This library enables dataflow computing on top of UPC++[8], another library that implements and extends in C++ the Unified Parallel C (UPC)[3] language.

UPC++ DepSpawn includes a large number of optimizations in order to provide competitive performance. One of the main mechanisms it uses to achieve this is a software cache managed by each process whose purpose is to minimize as much as possible the communications with other processes, since they are one of the most expensive operations in a distributed memory environment. An important problem however with this optimization is that the user is in charge of specifying the amount of memory devoted to this cache. This parameter is critical, since a cache that is too small will lead to much communications among the processes, and thus very poor performance, while a large cache will result in a poor usage of the memory available per process, particularly when it is scarce. Requiring users to provide the required cache size has the obvious shortcoming that they may ignore which may be an adequate value for it, thus requiring some manual tuning to find it. A related problem is that depending on the algorithm and/or input considered, the amount of processes used and the distribution of the data among the processes, the cache size required may not be uniform across the processes involved, and furthermore, it could evolve during the execution of the program. For these reasons, we developed an adaptive strategy for the cache size that independently establishes an adequate size in every process participating in a UPC++ DepSpawn execution without requiring any user intervention. In this paper we present the details of the operation and the implementation of the UPC++ DepSpawn cache and we describe and evaluate our cache auto-tuning strategy.

The rest of this manuscript is organized as follows. UPC++ and the UPC++ DepSpawn dataflow library are introduced in the next section. This is followed by an explanation of the functioning of the runtime software cache in Section 3. Then, Section 4 discusses the cache size problem and the proposed auto-tuning approach, which is evaluated in Section 5. A review of related work can be found in Section 6, while our conclusions and future work conclude this manuscript.

2 | UPC++ DEPSPAWN

Since UPC++ DepSpawn operates on the UPCC++ environment, a short introduction to the latter is needed before describing this dataflow computing library. UPC++[8] is a C++ library that provides the semantics of UPC[3] and some additional mechanisms by means of a rich C++ API. Its parallel executions consist of several processes that have both a traditional private memory and a shared global memory that all the processes can access, but which is partitioned among them. When a piece of data is stored in a region of the shared space that is local to a given process, such process can access it much faster than the other processes. While the variables of standard data types are stored in the private memory of each process, the shared memory stores only variables that belong to data types provided by UPC++. Namely, `shared_var<T>` represents a scalar variable of type `T` located in this shared space and `shared_array<T, B>` stands for an array of elements of type `T` that is distributed among the processes in the shared space following a block-cyclic fashion, the block size being `B`. This latter parameter is optional, its default value being 1. Other related UPC++ data types are `global_ptr<T>` and `global_ref<T>`, which generalize the concept of pointer and reference to items of type `T` located in the shared space, respectively.

The availability of the shared space and the one-sided communications it leads to due to the accesses to the items located in it from the different processes are the main component of UPC++. However, the library provides many other facilities, such as remote function invocation, that will not be described here for space reasons and also because they are not required to understand UPC++ DepSpawn.

As in UPC, UPC++ applications run in an SPMD style guided by the unique id of each process, and the synchronizations rely on traditional barriers and locks as well as new mechanisms such as futures or events. Still, as discussed in [13], these elements do not suffice to allow to efficiently express computations with irregular and/or complex patterns of dependences among their tasks. This way, a very simple dataflow approach for UPC++ was proposed based on DepSpawn[10], a library for dataflow computing in shared memory, so that the proposal was called UPC++ DepSpawn. It deserves to be mentioned that a new version of UPC++ has been proposed recently[14], but it still does not anything similar to the dataflow approach provided by UPC++ DepSpawn . Building a new version of this library adapted to this new UPC++ version is future work.

Just as in DepSpawn, the parallelization of an algorithm in UPC++ DepSpawn requires encapsulating each task as a function that only communicates with other tasks by means of its arguments. The type of each formal parameter of a function is used by the library to infer the nature of the usage of the associated argument. This way, since functions cannot modify arguments received by value or constant reference (`global_ref<const T>` in the case of UPC++ shared objects), arguments that correspond to such kinds of parameters are considered as only inputs to the function. On the contrary, since arguments received by means of non-constant references can be modified inside a function, the library considers them both as inputs and outputs of the task associated to the function. In UPC++ DepSpawn this latter kind of arguments will have types `global_ref<T>`. The library supports any kind of function, from C++11 lambda functions to class member functions, including of course regular C-like functions.

Once we have written an algorithm in terms of function calls with the proper arguments, the next step to parallelize it with this library is to rewrite each invocation `f(arg1, arg2, ...)` as `upcxx_depspawn(f, arg1, arg2, ...)` so that it will be analyzed and triggered by the library when its dependences are fulfilled. The library also provides the function `upcxx_wait_for_all()`, which provides synchronization points in which the application waits for all the pending tasks to complete.

The programming style associated to this proposal is illustrated in Listing 1, which shows the UPC++ DepSpawn version of a task-parallel Cholesky factorization performed by tiles of a given type `Tile`. It includes a sketch for function `dgemm` that shows that the first parameter is a non-constant reference to a shared object, which can be thus modified by the function, while the other arguments are received by value, so that `dgemm`

```
shared_array<Tile, 1> A(N * N);
#define _(i, j) ((i) * N + (j))
...

void dgemm(global_ref<Tile> dest, const Tile a, const Tile b) {
  // Implements dest = dest + a x b
}

for(i = 0; i < dim; i++) {
  upcxx_spawn(potrf, A[_(i,i)]);
  for(r = i+1; r < dim; r++) {
    upcxx_spawn(trsm, A[_(i,i)], A[_(r,i)]);
  }
  for(j = i+1; j < dim; j++) {
    upcxx_spawn(dsyrk, A[_(j,i)], A[_(j,j)]);
    for(r = j+1; r < dim; r++) {
      upcxx_spawn(dgemm, A[_(r,j)]), A[_(r,i)]), A[_(j,i)]));
    }
  }
}

upcxx_wait_for_all();
```

Listing 1 UPC++ DepSpawn example

cannot change their values if they are used by other tasks. As a result the runtime will infer that the first argument can be both read and written, while the remaining ones are only inputs. A complication generated by UPC++ is that its shared arrays are unidimensional. As a result, matrix `A`, which has `N`×`N` tiles, has to be declared as a vector of `N*N` tiles in which the accesses are linearized by means of the macro '_'.

As we can see, the code looks very much like the sequential counterpart, making very easy its development. Since UPC++ runs in SPMD its applications, all the processes execute this code. This allows all of them to know which is the sequence of tasks to execute as well as the dependences for each task. Each process chooses which tasks to execute following basically an owner computes rule based on the task arguments and their usage. Since the assignment rule and all the tasks and arguments are known by all the processes, they also know who is responsible for computing each dependence they may have as well as who they should notify because of dependences originated by tasks assigned to them.

The runtime of UPC++ DepSpawn includes a large set of optimizations in order to provide the best possible performance. For example, it automatically provides thread-level parallelism inside each UPC++ process on top of Intel Threading Building Blocks[15], so that each process can concurrently execute multiple tasks assigned to it, always respecting the dependences inferred from the code. This also has other benefits, as for example the cost of the construction of the task dependency graph is amortized among all the threads of the same process, as they share it. This manuscript is focused though on another enhancement, namely the management of a software cache in each process. We discuss its operation and implementation in the next section, and we propose an auto-tuning strategy for its size in Section 4.

3 | THE SOFTWARE CACHE

As mentioned previously, the runtime of UPC++ DepSpawn provides a software-managed cache to each process. Its purpose is to minimize the communications between processes, as they are one of the most expensive operations in a UPC++ DepSpawn execution given the distributed memory of the environment supported. We will first discuss how to use this cache and then we will explain how it works internally.

It was decided not to change the way that every UPC++ access to global shared data works for two reasons. First, the purpose was to accelerate the execution of the UPC++ DepSpawn codes without affecting the way the rest of the UPC++ application works. Second, the library was designed so that it can be used on top of any existing UPC++ installation without modifying it. This way, rather than involving the cache in the accesses of any UPC++ standard global reference `global_ref<T>`, a separate type `cached_global_ref<T>` was designed which does interact with, and thus benefits from, the software cache. The basic idea is to use this type in the formal parameters of the spawned functions so that the parameters of this type provide cached access to the associated element of the global shared space to which they are associated.

This new data type is constructed from a standard `global_ref<T>` and it can be used following either a manual or an automatic approach. Under the manual approach, programmers change by hand the types of the formal parameters of the functions that constitute the UPC++ DepSpawn tasks that they think that may benefit from the cache, replacing them by the new `cached_global_ref<T>` type that accesses the cache. These parameters are easy to identify because the cache only benefits the accesses to remote data and since, as explained in Section 2, the runtime follows an owner computes rule, then the read-only inputs are the ones who are likely to be remote, and thus be a proper target for the software cache. Under the automatic approach however, users do not need to make change. Rather, they just need to compile the program using the macro `UPCXX_DEPSPAWN_AUTOMATIC_CACHE`. This compile-time flag orders the library to wrap every user-provided original function used in a `upcxx_spawn` call inside a new function that uses the `cached_global_ref<T>` type in its formal parameters whenever read-only remote accesses are identified in the associated argument. This achieves analogous results to the manual option with less programmer effort at the cost of a negligible overhead and less control.

As for the internal behavior of the implementation, whenever a `cached_global_ref<T>` is created from a `global_ref<T>`, the following steps take place:

1. First the runtime checks whether the associated element happens to be located in the part of the shared space with affinity to the process, which means that it is stored in its local memory. If this is the case, the cache is by passed and the `cached_global_ref<T>` just directly provides access to the original element in the local memory.

2. Otherwise, the cache is searched for a copy of this element. If present, the `cached_global_ref<T>` points to the existing copy.

3. If no copy is found, space is allocated in the cache to hold a local copy, and a remote access is performed to bring the element from the remote memory where it resides. Once it arrives, the `cached_global_ref<T>` provides access to the local copy just created in the cache.

The contents of the cache are managed following a LRU replacement strategy, a maximum cache size and an allowed slack above this maximum size. Namely, when upon a new insertion the cache size becomes larger than the maximum size plus the slack, the runtime analyzes the elements in the cache starting at the bottom on the LRU stack and progressing towards the top, freeing during this examination those elements that are not in use. This process stops when either the cache size is again the maximum one allowed or the top is reached, whatever happens first. In order to detect whether a cache element is in use, each cache item has a counter of the number of existing `cached_global_ref<T>` objects that point to it. When this counter, which is managed using atomic operations, is zero, the item can be freely removed from the cache if desired.

The two parameters mentioned above can be configured in this cache. First, its size is specified by the user either by means of an environment variable called `UPCXX_DEPSPAWN_MAX_CACHE` or an explicit invocation in the application to a runtime configuration function called `set_UPCXX_DEPSPAWN_MAX_CACHE`. As for the slack allowed beyond this maximum size before triggering a cleaning process to reduce the number of elements of the cache to this maximum, its value is provided by the environment variable `UPCXX_DEPSPAWN_SLACK_CACHE` or the runtime function `set_UPCXX_DEPSPAWN_SLACK_CACHE`. The motivation for the slack in the implementation, which in a traditional cache would be zero, is to reduce the overhead of the cache size reduction operations. The reason is that they involve a global locking of the cache, which can be slightly bad for performance. The usage of a small slack allows to keep basically the same maximum cache size, while noticeably reducing the number of cache size reduction operations and thus the cache management overhead.

It deserves to be mentioned that this cache is also designed with coherency in mind, as its management includes mechanisms to evict those elements that have become stale because they have been modified by another process in the meantime. This is possible thanks to the fact that, as explained in [13], every UPC++ DepSpawn process has a full view of the task dependency graph (TDG). In fact, each process knowns about every task in the program, including where it is run and what it reads and writes. This way, whenever a process learns that a task T has been completed, it evicts from its local cache those pieces of data that belonged to the writing set of T, as they will be no longer valid. Notice that this eviction is not needed before the writing task T is known to have finished. The reason is that if there were any access to the data within the writing set of T by any subsequent local task L, this would imply a dependency, and a result, L could only begin its execution once its owner process is notified that T finished. What is critical is that the cache eviction takes place before L begins its execution, which is the case in our runtime.

The implementation of the cache is depicted in Figure 1. The cache storage is provided by the doubly-linked list at the right of the figure. The elements of this list are ordered from the most recently used, at the beginning of the list, and the LRU element, which is at the bottom. The reordering associated to an access to an element that is not the topmost one is quickly performed by just relinking the element used at the beginning of the list, which is a matter of changing just a few pointers. Searching for an element in a list of size N is a sequential process with complexity $\mathcal{O}(n)$, which is very slow. For this reason, this list only provides the storage for the cache elements and a few related data such as the counter of the number of uses mentioned before, the size of the element stored and a flag indicating whether the element has been written or not, but it does not hold the associated global address. This information is stored in the directory of the cache, implemented as a traditional hash table

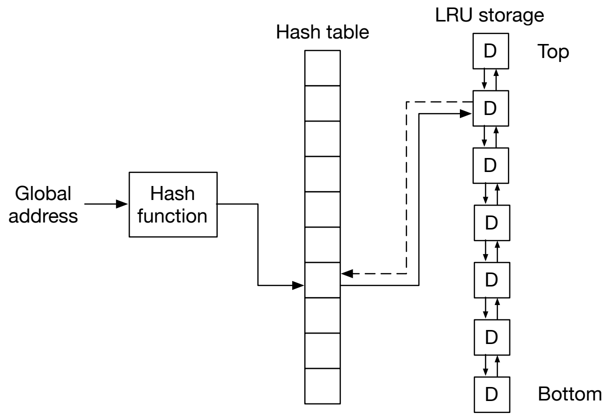

FIGURE 1 Internal structure of the software cache

with support for collisions shown at the left of Figure 1. This table is indexed with the global address stored in the global reference and it provides a pointer to the underlying storage of the associated cached element in the storage on a hit. The table is dimensioned so that the complexity is the search is usually $\mathcal{O}(1)$. We can see in the figure that the list elements also hold a pointer back to their associated entry in the hash table (dashed line). This pointer is needed during the cleaning processes that reduce the cache size, since the removal of the bottommost elements of the storage also involves naturally the removal of their associated entries in the hash table, which can be quickly found through these pointers.

Finally, the implementation is also highly optimized from the point of view of parallelism, so that the multiple threads that a UPC++ DepSpawn program may be using to run the tasks assigned to it can operate on parallel on the cache as much as possible. For example, while a thread remains blocked during the handling of a miss in the cache, the cache can be simultaneously accessed by all the other threads, and in fact it uses an optimized synchronization strategy in order to maximize the concurrency on its operation. Namely, it not only distinguishes between shared/read-only locking and exclusive/write locking, but it also applies fine grain locking whenever possible.

4 | A CACHE SIZE AUTO-TUNING STRATEGY

Since remote accesses are by far the most expensive operation in a distributed memory environment, it is critical to provide a cache large enough to minimize them. Simultaneously, oversizing the software cache leads to a poor usage of the memory of each process, particularly in situations in which it may be scarce. Also, the best cache size could not be the same for different processes, and in general it is difficult to estimate it in advance. Furthermore, the optimal cache size could evolve dynamically during the execution, as different stages usually have different locality and access patterns. This way, given the capabilities described in the previous section, it is very difficult for users to identify the best cache size in each process in each moment and thus obtain the best possible behavior. This led us to develop an auto-tuning algorithm that enables the runtime to adjust dynamically the cache size.

We designed our auto-tuning algorithm so that it is invoked periodically. This way, it takes its decisions depending on the behavior observed during one or several periods rather than on individual accesses. Namely, every `TunePeriod` consecutive accesses to the cache a period is completed and the auto-tuning function is invoked. The invocation is performed in a thread-safe fashion by the thread that runs the task responsible for this access. This way, no additional management threads are needed by our approach.

Our algorithm can be in one of two different states:

- Sometimes the algorithm needs to estimate how much of the cache is actually in use. This is achieved in the *deepest_hit_evaluation* state, which computes the deepest hit during each period, that is, the furthest hit from the top of the LRU stack. Since the cache is fully associative, this estimates the maximum number of elements in use during the period considered.
- When not in that state, we say that the algorithm is in *standard* state. In this case, the algorithm takes decisions based on the hit rate on the cache.

The algorithm, which is shown in Listing 2, receives as input in each invocation the total number of `hits` accumulated in the cache. In addition, the function has an internal state consisting of a few persistent variables. For example, `last_hits` keeps the value of `hits` in the previous

```
void periodic_tune(hits) {
   new_period_hits = hits - last_hits;
   if (deepest_hit_evaluation) {
    if(++periods_counter == periods_limit) {
      if (deepest_hit > prev_deepest_hit) {
        prev_deepest_hit = deepest_hit; // reevaluate
        periods_limit = max(1, (size - deepest_hit) / (3 * TunePeriod));
      } else {
        max_size = (deepest_hit + size) / 2;
        deepest_hit_evaluation = false;
      }
      periods_counter = 0;
    }
   } else {
     hit_rate = new_period_hits / (float)TunePeriod;
     if (hit_rate > 0.98) {
       trigger_deepest_hit_evaluation();
     } else if (hit_rate < 0.96) {
       if (max_size < min_items) {
         proposed_size = max_size + TunePeriod;
       } else {
         rate = (TunePeriod - new_period_hits) / TunePeriod;
         proposed_size = max_size + max_size_* rate;
         if (new_period_hits < (last_period_hits + 0.05*TunePeriod)) {
           if (++periods_counter == 4) {
             trigger_deepest_hit_evaluation();
           }
         } else {
           periods_counter = 0;
         }
       }
       max_size = min(proposed_size, max_items);
     }
  }
  last_period_hits = new_period_hits;
  last_hits = hits;
}

void trigger_deepest_hit_evaluation() {
  deepest_hit_evaluation = true;
  periods_counter = 0;
  periods_limit = max(1, size / (3 * TunePeriod));
  deepest_hit = prev_deepest_hit = 0;
}
```

Listing 2 Cache auto-tuning algorithm

invocation of the algorithm, as seen in line 36. This way, by subtracting it from `hits` in line 2, we obtain the number `new_period_hits` of hits during the latest `TunePeriod` accesses. Another example is the boolean `deepest_hit_evaluation`, which indicates whether the algorithm is in the *deepest_hit_evaluation* state.

Let us now explain how the *deepest_hit_evaluation* state, implemented in lines 4-13 proceeds. Part of the work of this state takes place out of the algorithm itself, as in this state the cache computes on every hit the variable `deepest_hit`, which represents the maximum distance from the top of the LRU stack of any hit in the cache. The algorithm allows for a varying number `periods_limit` of `periodic_tune` invocations to perform this analysis. The initial value, set in the function `trigger_deepest_hit_evaluation` (lines 39-44) used to change from the *standard* to the *deepest_hit_evaluation* state, depends on the current cache size `size` and the `TunePeriod`, as seen in line 42.

When `periods_limit` is reached and `deepest_hit` turns out to be deeper than the depth `prev_deepest_hit` recorded in the preceding periods of evaluation, the algorithm reschedules in lines 6-7 another round of periods of analysis whose number depends on the cache size and the `deepest_hit` recorded. When this is not the case, the maximum size allowed for the cache `max_size`, which the element tuned by our algorithm, is trimmed to `(deepest_hit + size) / 2` and the algorithm leaves this mode in lines 9-10.

When the algorithm is in *standard* mode, in lines 15-33, its behavior depends on the hit rate during the last period, which is computed in line 15. The algorithm assumes that a hit rate above 98% means that it is worthwhile to examine whether the cache size could be reduced, and thus it starts the deepest hit evaluation process described above in line 17. When the hit rate is between 96% and 98% our algorithm assumes that the situation is good enough for the performance of the cache, but not so good as to try to save memory space, so no new measures are taken. Finally, if the hit rate falls below 96% the algorithm tries to enhance the behavior of the cache adjusting its size between two limits. The lower one is a minimum `min_items`, which is assumed to be small enough to not be considered a meaningful waste of memory, while the upper one is a maximum `max_items` that is estimated as an upper limit for the cache size. These limits are currently associated to caches sizes of 500MB and 4GB, respectively.

The growth of the cache size is not uniform while the minimum desired hit rate is not increased. Namely, as we can see in lines 19-20, as long as the maximum cache size allowed `max_size` does not reach the `min_items` minimum, the new proposed maximum size `proposed_size` grows in increases of `TunePeriod` elements. The reason is that this is the number of accesses to serve before the next tuning evaluation and we are willing to devote new storage for each one of those accesses if it turns out to miss in the cache while the cache is below this reasonable minimum `min_items` storage.

After that point we take a bayesian approach in which the maximum proposed size of the cache grows by a ratio given by the miss rate computed in line 22 so that the worse the cache behaves the faster it grows. In this process, the algorithm examines in line 24 whether the number of hits per period is actually growing with the growth of the cache beyond a noise rate estimated at 5% of the accesses. When this lack of meaningful improvement has happened for 4 consecutive periods, the algorithm enters a period of evaluation of the cache hit depth in line 26. This seeks to reassess the cache size because this low improvement of the hit rate hints that this stage of the algorithm may be lacking locality, and as a result the growth of the cache may have been useless.

As a final step in the *standard* mode, the `proposed_size` computed in the previous steps can become the new actual `max_size` as long as it does not exceed `max_items` in line 32.

5 | EVALUATION

We have performed our experiments in a cluster with 32 nodes with 24 cores each, as reflected in Table 1, which describes the most important characteristics of the hardware and software involved. In addition, all the codes were compiled with the optimization level O3. All the executions used a single UPC++ process per node, and this process had 24 different threads managed by the Intel TBB, so that there was exactly one thread per core in each node.

As benchmarks we used the right-looking Cholesky decomposition of a lower triangular matrix used as example in Listing 1 and a LU decomposition, whose algorithm is depicted in Listing 3. Both are important kernels with complex access patterns that benefit from the application of the dataflow approach and are thus often used for evaluating this kind of runtimes [9,10,11,13]. Our implementations rely on the highly optimized OpenBLAS library version 0.3.9 for the BLAS kernels used by these algorithms. For each algorithm we considered two problem sizes, namely a matrix of 40000×40000 double-precision floating point elements and another one of 70000×70000. Also, we followed the same experimental methodology as in [13], where for each algorithm, matrix size and number of nodes, we run experiments combining different tile sizes between 100×100 and 500×500, noticing that the optimal tile sizes were in that range, with all the possible definitions of the grid of nodes associated to a number of nodes. Finally, the combination of tile size and node grid that achieved the best performance was chosen for the experiments associated with that algorithm, matrix size and number of nodes.

TABLE 1 Experimental environment.

Feature	Value
#Nodes	32
CPUs/Node	2 x Intel Xeon E5-2680 v3
CPU Family	Haswell
CPU Frequency	2.5 GHz
#cores/CPU	12
Total #cores	$32 \times 2 \times 12 = 768$
Memory/node	128GB DDR4
Network	Infiniband FDR@56Gbps
Compiler	g++ 6.4
GASNet	1.28.0
Intel® TBB	2018.6.274

```
for (int k = 0; k < N; k++) {
  lu0(A[k][k]);
  for (int j = k+1; j < N; j++)
    fwd(A[k][k], A[k][j]);
  for (int i = k+1; i < N; i++) {
    bdiv(A[k][k], A[i][k]);
    for (int j = k+1; j < N; j++) {
      bmod(A[i][k], A[k][j], A[i][j]);
    }
  }
}
```

Listing 3 LU decomposition

A first point of interest is to observe the dynamic evolution of the cache size during the execution of the algorithms when our auto-tuning proposal is used. Figure 2 shows this evolution during the execution of the two problem sizes considered when using 4 and 16 nodes for the Cholesky algorithm and and Figure 3 shows the results for the LU decomposition. Since each process may have a different cache size, the figures plot the average at each point. Notice that the number of accesses to the caches (x axis) varies with (a) the problem size, (b) the tile size, as the larger the tiles, the smaller the number of tasks and accesses and (c) the number of nodes, as the more nodes, the fewer tasks but also the larger in general the ratio of remote accesses. As one can see from the codes in Listings 1 and 3, as the outer loop of the algorithm proceeds, the amount of work and thus the number of reuses per remote tile is reduced. This way the cache size is relatively stable for most of the execution, while high hit rates are achieved because there is much reuse per remote element. Then, in the last stages of the algorithm there are increasingly few reuses of the brought tiles. As a result, the algorithm scrambles to increase the hit rate allowing a quick growth of the cache size, as it is still much below the maximum permitted and our maximum priority is to increase the hit rate in order to reduce the runtime.

Let us now turn our attention to the comparison between a manual approach and our automated proposal. After some hand tuning, we found that we could reach optimal cache hit rates using fixed size caches of one thousand tiles for the first problem size and one thousand five hundred for the second one. This configuration allows each cache to store about two rows and two columns of tiles of these matrices. As for the memory footprint of these caches, it is between 320MB and 2GB, which seems reasonable. We then run the same tests using the auto-tuning algorithm and compared the cache behavior of the manual and the automated cache configuration in Figure 4, which shows the reduction of the hit rate of the automated approach with respect to the manual one. We find that the maximum impact is smaller than 1%, while in most experiments it does not reach even 0.5%. In fact the average hit rate reduction across all the experiments is just 0.27%. This good behavior of our auto-tuning approach means that the cache with automated management did not give place to meaningful runtime overheads. This way, our Cholesky and LU

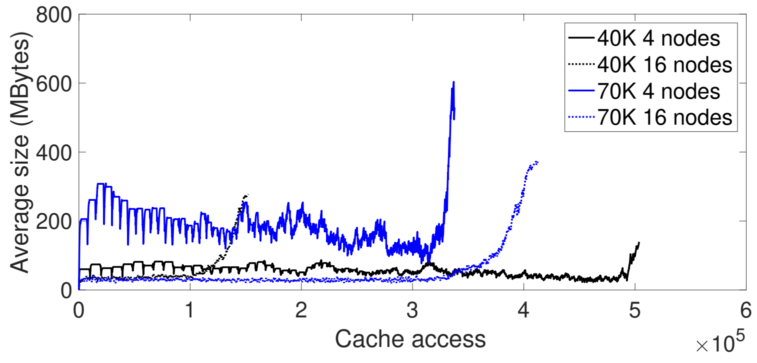

FIGURE 2 Evolution of the cache size in Cholesky when using auto-tuning

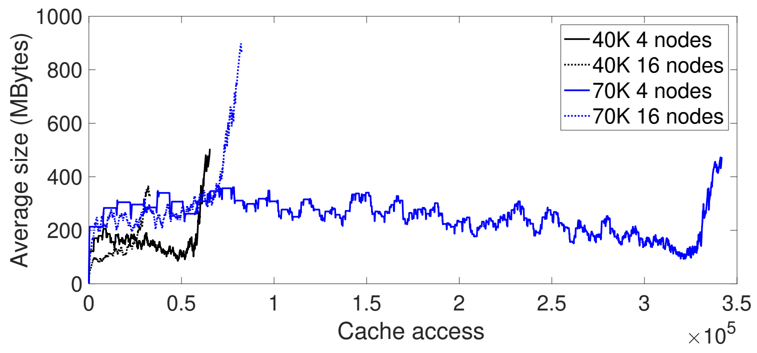

FIGURE 3 Evolution of the cache size in LU when using auto-tuning

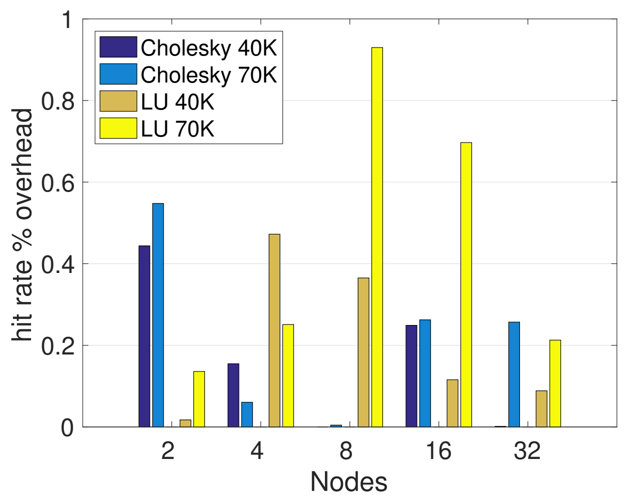

FIGURE 4 Hit rate overhead (reduction) when using auto-tuning

benchmarks with automated cache size management were on average just 1.1% and 1.7% slower than the baselines with cache sizes manually specified by the user, respectively.

Regarding the memory consumption of the auto-tuning algorithm, it is in fact usually smaller than that of the cache size we conservatively estimated to make sure the working set of the applications fit in the software cache. Namely, when the application finished, the size of the cache with automatic size management was on average 13% and 66% smaller than the one used with the manual configuration in the executions of the Cholesky decomposition on the small and the large matrix, respectively. The reductions observed for the execution of the LU decomposition on the small and the large problem size grew to 69% and 78%, respectively. Let us remember that in some of its stages our auto-tuning algorithm considers that there is some minimum amount of memory that is always fine to devote in the cache, even if it is underused, and that it tries to be conservative in its evolutions, as wasting some memory has usually much less severe effects on performance than triggering an increase in communications due to a too small cache. This is an important reason why the memory footprint reduction for the small problems is smaller than for the large problems.

6 | RELATED WORK

There have been several proposals to enable the effective development of task parallel algorithms with complex patterns of dependencies on distributed memory systems. Some of them have been ad-hoc implementations addressing a single problem, such as the Cholesky[16] or the LU factorizations[17], the latter one being implemented on top of the UPC language.

As for the generic approaches, they can be classified in two large groups depending on whether the dependencies are explicitly or implicitly provided. A representative example of the first group is the Parameterized Task Graph (PTG) model[18], which is the basis for the PaRSEC framework[19]. In this model the user expresses the dependencies between tasks using a domain specific language from which a code that supports the dataflow parallel execution of the algorithm expressed is statically generated. The availability of the task graph in advance enables this approach to apply more ambitious optimizations than the dynamic approaches, the obvious shortcomings being the impossibility to express applications whose dependencies can only be known at runtime and the difficulty that the discovery and correct conveyance of the dependencies may entail for the user. Other explicit approaches discover the dependencies during the execution with the help of program objects that the programmer uses to express such dependencies. This is the case of the UPC++ events discussed in Section 2. They are inspired by and share the same limitations as those of Phalanx[20], a programming model for distributed heterogeneous machines whose synchronization relies on them when barriers do not suffice. Single assignment variables[21,22] and futures, these latter ones having been also discussed in Section 2, offer some degree of implicitness but they are only a natural option for read-after-write data dependencies, and even for the codes where these are the only dependencies to consider, their effective use to express optimized dataflow computations requires additional mechanisms. The reason is that in a dataflow model tasks should start their execution only when their data are ready, while these approaches require the dependent tasks to either poll or block on them until they are ready, i.e., until the dependencies they are associated to are fulfilled. The additional mechanisms needed must thus be able to associate each task with the set of dependence-carrying items it depends on, and to ensure that the task will only be launched for execution when all those futures or synchronization variables are ready. This is the case of the dataflow objects provided by HPX[23], which supports task based programming on distributed systems on top of C++, or the combination of distributed data-driven futures and a `await` clauses for asynchronous tasks in[24], which provide a similar functionality in Habanero-C MPI. Chapel's synchronization variables[21] present similar restrictions with the difference that they can be written multiple times at the cost that each write can only be read by a single synchronization access. Finally,[14] also provides mechanisms to chain the multiple futures on which a computation may depend and only launch it as a callback when they are ready.

The implicit approaches typically only require from the user the annotation of the inputs and the outputs of each task in an apparently sequential code, and sometimes also the registration of the data to use. These proposals identify the dependencies that must be fulfilled to guarantee sequential consistency by considering the data accessed by each task during the sequential order of execution of the tasks with the help of a runtime attached to the application, so that this identification happens during the execution. Some of them follow a centralized approach in which one process is responsible for the discovery of the dependencies and the management of the TDG, while the other participants act as servers that run the tasks assigned to them. This is the case of ClusterSs[25], which annotates Java code to indicate the usage (input, output or both) of each task parameter or OmpSs[26], which relies on compiler directives similar to OpenMP to provide this and other informations on C/C++ codes. The potential scalability limitations of this approach are ameliorated in[26] by supporting the submission of tasks that can can be further decomposed and parallelized within a node. A decentralized library-based approach is taken by StarPU[9], which relies on handles to pre-registered data to express its dependencies and data distribution, MPI being used for the distributed executions, which take place on separate processes[11], differing from the transparent support of multithreading within each process in UPC++ DepSpawn, which reduces the communication and TDG handling overheads.

A proposal that supports task-based dataflow execution on distributed systems on top of a novel programming paradigm is Legion[27], which identifies logical regions that can be accessed by different tasks and which are decoupled from the different physical regions to which they may be mapped. The large flexibility of the model and its new concepts requires a non-trivial programming on top of a C++ library API, which led to the development of a new language and compiler[28] that substantially simplify the exploitation of Legion. Another language-based proposal that provides dataflow parallelism on distributed systems is Swift/T[29], which relies on MPI to glue user codes written in traditional languages. A distinctive property of Swift/T is its purely functional nature, which contrary to most proposals, including the one discussed in this paper, does not allow tasks to modify their arguments.

Finally, HabaneroUPC++[30] has in common with UPC++ DepSpawn that it extends UPC++ in order to provide a compiler-free library that enhances the usability of this PGAS environment, in this case by supporting rich features for dynamic task parallelism within each process. Namely, it relies on UPC++ for PGAS communication and RPC, and Habanero-C++ for supporting intra-place work-stealing integrated with function shipping.

The contributions in this paper are also related auto-tuning. While there is a large body of work about the auto-tuning of user applications and kernels as well as hardware devices following very different approaches, this is not the case about the automatic adaptation of internal elements of runtimes. There are however some works in the area of self-adaptivity for software managed caches. This is the case of[31], which adapts the coherence mechanism for a cache of this kind. More related is[32], which dynamically adapts the distribution of space inside caches with one region devoted to new items and another one assigned to the frequently use items, and the parameters of the rules that decide when to move one item from one region to the other one. Our work rather focuses on a single-region fast access cache heavily optimized for multithreaded usage in which the total size of the cache is the element to be tuned.

7 | CONCLUSIONS

As far we know, UPC++ DepSpawn is the first library to provide dataflow computing on a PGAS environment. Another outstanding characteristic is that it provides a very terse an intuitive notation based on the data types of each task formal parameters and the sequential specification of the algorithm to implement. Furthermore, it allows to exploit parallelism both among distributed memory process and also among threads within each process that are automatically managed by its runtime.

This paper has presented in detail one of the most important optimizations of the library, namely a software cache that can strongly reduce communications by allowing to reuse data brought from remote memories. We have also dealt with a limitation of the first proposal of the library in[13], namely the need to specify a cache size for each process in order to reduce the communications with other processes. This value is critical, as communications are very expensive operations, while memory can often be a scarce resource in many scenarios. Relatedly, it can be very difficult for users to assess the amount of cache that a given algorithm, number of processors, data distribution strategy, and input to process may need; and furthermore this value can evolve during the execution of the algorithm. As a result, the automated approach introduced in this paper in which the runtime dynamically estimates the amount of cache it needs is a great improvement to UPC++ DepSpawn. Our evaluation has shown that the algorithm proposed reduces on average only by 0.27% the hit rate of the cache with respect to an optimal configuration, the maximum reduction being bellow 1%. These cache hit reductions resulted in average runtime growths below 2%, which we deem very adequate. Relatedly, the proposed cache size auto-tuning algorithm proved to adjust the memory consumption, providing important savings with respect to a conservative manual estimation.

As future work we plan to continue improving the performance of UPC++ DepSpawn. We would also like to explore its integration on top of the UPC++ version proposed in[14] and develop a version that supports heterogeneous computing, so that tasks can also be run in hardware accelerators such as GPUs.

ACKNOWLEDGMENTS

This research was funded by the Ministry of Science and Innovation of Spain (TIN2016-75845-P and PID2019-104184RB-I00, AEI/FEDER/EU, 10.13039/501100011033), and by the Xunta de Galicia co-funded by the European Regional Development Fund (ERDF) under the Consolidation Programme of Competitive Reference Groups (ED431C 2017/04). We acknowledge also the support from the Centro Singular de Investigación de Galicia "CITIC", funded by Xunta de Galicia and the European Union (European Regional Development Fund- Galicia 2014-2020 Program), by grant ED431G 2019/01. We also acknowledge the Centro de Supercomputación de Galicia (CESGA) for the use of its computers.

Conflict of interest

The authors declare no potential conflict of interests.

References

1. Yelick K, Bonachea D, Chen WY, et al. Productivity and Performance Using Partitioned Global Address Space Languages. In: *Proc. 2007 International Workshop on Parallel Symbolic Computation*. PASCO'07. ; 2007: 24–32

2. Numrich RW, Reid J. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum* 1998; 17(2): 1–31. doi: 10.1145/289918.289920

3. Burke MG, Knobe K, Newton R, Sarkar V. UPC language specifications, v1.2. Tech. Rep. LBNL-59208, Lawrence Berkeley National Lab; 2005.

4. Charles P, Grothoff C, Saraswat V, et al. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In: *20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. ; 2005: 519-538.

5. Chamberlain B, Callahan D, Zima H. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.* 2007; 21(3): 291-312. doi: 10.1177/1094342007078442

6. Yelick KA, Graham SL, Hilfinger PN, et al. Titanium. In: *Encyclopedia of Parallel Computing*. Springer US. 2011 (pp. 2049–2055)

7. Nieplocha J, Palmer B, Tipparaju V, Krishnan M, Trease H, Aprà E. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *International J. of High Performance Computing Applications* 2006; 20(2): 203–231. doi: 10.1177/1094342006064503

8. Zheng Y, Kamil A, Driscoll MB, Shan H, Yelick K. UPC++: A PGAS Extension for C++. In: *IEEE 28th International Parallel and Distributed Processing Symposium*. IPDPS 2014. ; 2014: 1105-1114

9. Augonnet C, Thibault S, Namyst R, Wacrenier P. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 2011; 23(2): 187-198. doi: 10.1002/cpe.1631

10. González CH, Fraguela BB. A framework for argument-based task synchronization with automatic detection of dependencies. *Parallel Computing* 2013; 39(9): 475-489. doi: 10.1016/j.parco.2013.04.012

11. Agullo E, Aumage O, Faverge M, et al. Harnessing clusters of hybrid nodes with a sequential task-based programming model. In: *International Workshop on Parallel Matrix Algorithms and Applications*. PMAA 2014. ; 2014.

12. Fraguela BB. A Comparison of Task Parallel Frameworks based on Implicit Dependencies in Multi-core Environments. In: *50th Hawaii International Conference on System Sciences*. HICSS'50. ; 2017: 6202-6211

13. Fraguela BB, Andrade D. Easy Dataflow Programming in Clusters with UPC++ DepSpawn. *IEEE Transactions on Parallel and Distributed Systems* 2019; 30(6): 1267-1282. doi: 10.1109/TPDS.2018.2884716

14. Bachan J, Baden SB, Hofmeyr S, et al. UPC++: A High-Performance Communication Framework for Asynchronous Computation. In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. ; 2019: 963-973.

15. Reinders J. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly. 1 ed. 2007.

16. Gustavson FG, Karlsson L, Kågström B. Distributed SBP Cholesky Factorization Algorithms with Near-optimal Scheduling. *ACM Trans. Math. Softw.* 2009; 36(2): 11:1–11:25. doi: 10.1145/1499096.1499100

17. Husbands P, Yelick K. Multi-threading and One-sided Communication in Parallel LU Factorization. In: *2007 ACM/IEEE Conference on Supercomputing*. SC'07. ; 2007: 31:1–31:10

18. Cosnard M, Loi M. Automatic task graph generation techniques. In: *28th Annual Hawaii International Conference on System Sciences*. . 2 of *HICSS'28*. ; 1995: 113-122 vol.2

19. Danalis A, Jagode H, Bosilca G, Dongarra J. PaRSEC in Practice: Optimizing a Legacy Chemistry Application through Distributed Task-Based Execution. In: *2015 IEEE International Conference on Cluster Computing*. ; 2015: 304-313

20. Garland M, Kudlur M, Zheng Y. Designing a unified programming model for heterogeneous machines. In: *2012 International Conference on High Performance Computing, Networking, Storage and Analysis*. SC'12. IEEE; 2012: 67:1–67:11

21. Cray Inc r. Chapel Language Specification Version 0.984. 2017.

22. Breitbart J. A dataflow-like programming model for future hybrid clusters. *International J. of Networking and Computing* 2013; 3(1): 15–36.

23. Kaiser H, Heller T, Adelstein-Lelbach B, Serio A, Fey D. HPX: A task based programming model in a global address space. In: *8th International Conference on Partitioned Global Address Space Programming Models*. PGAS '14. ; 2014: 6:1–6:11

24. Chatterjee S, Tasirlar S, Budimlic Z, et al. Integrating Asynchronous Task Parallelism with MPI. In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IPDPS 2013. ; 2013: 712-725

25. Tejedor E, Farreras M, Grove D, Badia RM, Almasi G, Labarta J. A high-productivity task-based programming model for clusters. *Concurrency and Computation: Practice and Experience* 2012; 24(18): 2421–2448. doi: 10.1002/cpe.2831

26. Bueno J, Martorell X, Badia RM, Ayguadé E, Labarta J. Implementing OmpSs Support for Regions of Data in Architectures with Multiple Address Spaces. In: *27th International Conference on Supercomputing*. ICS '13. ; 2013: 359-368.

27. Bauer M, Treichler S, Slaughter E, Aiken A. Legion: Expressing Locality and Independence with Logical Regions. In: *International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. ; 2012: 66:1–66:11.

28. Slaughter E, Lee W, Treichler S, Bauer M, Aiken A. Regent: A High-productivity Programming Language for HPC with Logical Regions. In: *International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. ; 2015: 81:1–81:12

29. Wozniak JM, Armstrong TG, Wilde M, Katz DS, Lusk E, Foster IT. Swift/T: Large-Scale Application Composition via Distributed-Memory Dataflow Processing. In: *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. CCGrid 2013. ; 2013: 95-102

30. Kumar V, Zheng Y, Cavé V, Budimlić Z, Sarkar V. HabaneroUPC++: A Compiler-free PGAS Library. In: *8th International Conference on Partitioned Global Address Space Programming Models*. PGAS '14. Association for Computing Machinery; 2014; New York, NY, USA: 5:1–5:10

31. Bennett JK, Carter JB, Zwaenepoel W. Adaptive software cache management for distributed shared memory architectures. In: *17th Annual International Symposium on Computer Architecture (ISCA)*. ; 1990: 125-134.

32. Einziger G, Eytan O, Friedman R, Manes B. Adaptive Software Cache Management. In: *Proceedings of the 19th International Middleware Conference*. Middleware '18. Association for Computing Machinery; 2018; New York, NY, USA: 94–106

AUTHOR BIOGRAPHIES

Basilio B. Fraguela received the M.S. and the Ph.D. degrees in computer science from the Universidade da Coruña, Spain, in 1994 and 1999, respectively. Currently, he is a Professor in the Departamento de Enxeñaría de Computadores of the Universidade da Coruña, where he has been a Faculty Member since 1995. His primary research interests are in the fields of programmability, high performance computing, heterogeneous systems and code optimization. His homepage is http://gac.udc.es/~basilio

Diego Andrade received the M.S. and Ph.D. degrees in computer science from the Universidade da Coruña, A Coruña, Spain, in 2002 and 2007, respectively. He is currently an Associate Professor at the Departamento de Enxeñaría de Computadores of the Universidade da Coruña, where he has been a Faculty Member since 2006. His research interests focuses in the fields of performance evaluation and prediction, analytical modeling, and compiler transformations. His homepage is http://gac.udc.es/~diego